Programming Languages
Lecture15 – Scheme

남 범 석

bnam@skku.edu

Infix Notation

- We usually write algebraic expressions like this:

  a + b

- This is called **infix notation**, because the operator ("+") is inside the expression

- A problem is that we need parentheses or precedence rules to handle more complicated expressions:

  For Example :

  a + b * c = (a + b) * c ?

  = a + (b * c) ?

Infix, Postfix, & Prefix notation

- There is no reason we can't place the operator somewhere else.
- How ?
- Infix notation : a + b
- Prefix notation : + a b
- Postfix notation: a b +

Example

infix	*postfix*	*prefix*
(a + b) * c	a b + c *	* + a b c
a + (b * c)	a b c * +	+ a * b c

Infix form : <*identifier*> <*operator*> <*identifier*>

Postfix form : <*identifier*> <*identifier*> <*operator*>

Prefix form : <*operator*> <*identifier*> <*identifier*>

Practice: Infix to Prefix and Postfix

- *x + y*
- *(x + y) - z*
- *w * ((x + y) - z)*
- *(2 * a) / ((a + b) * (a - c))*

- *Prefix:*
  - *+ x y*
  - *- + x y z*
  - ** w - +x y z*
  - */*2 a *+a b - a c*

Lambda calculus and Scheme

- Functional programming is essentially an applied lambda calculus with built in
  - constant values
  - functions

- In Scheme, we use (* x x) for x*x instead of λx. x*x

Run, Scheme, Run

- $scheme

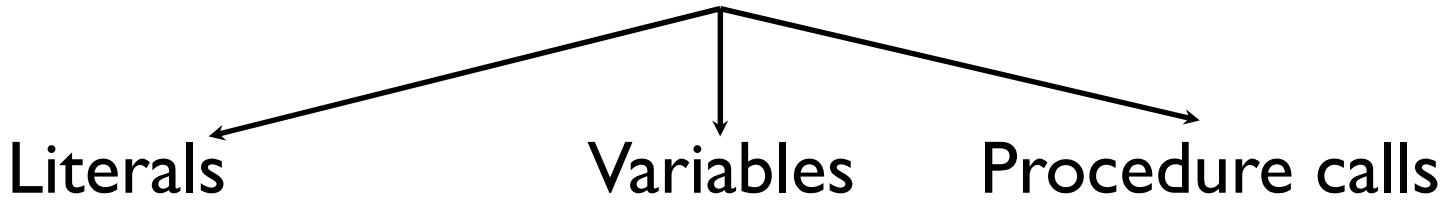
```
$scheme

MIT/GNU Scheme running under GNU/Linux
Type `^C' (control-C) followed by `H' to obtain
information about interrupts.
…
```

- In order to quit, press Ctrl+C and Q
- Press Ctrl+C twice to restart your program

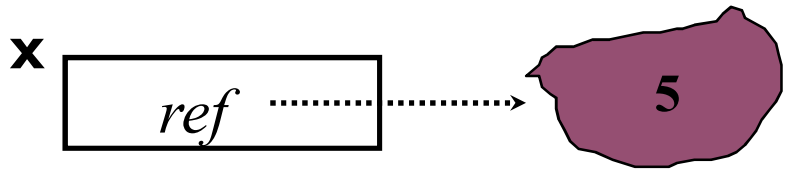
Interrupt Scheme interpreter

- More interrupt options
  - ^B: Enter a breakpoint loop.
  - ^L: Clear the screen.
  - D: Debugging: change interpreter flags.
  - E: Examine memory location.
  - H: Print simple information on interrupts.
  - I: Ignore interrupt request.
  - **Q: Quit instantly, killing Scheme.**
  - R: Hard reset, possibly killing Scheme in the process.
  - T: Stack trace.
  - …

Expressions



- Literals
  - 2
  - “abc”
  - #t
- Variables
  - Identifier *represents* a variable. Variable reference *denotes* the value of its binding.



Functions

- A function call looks like a list.

- Primitive arithmetic functions
  - `+ - * /`
    - `(+ 5 2)` yields 7
    - `(+ ( / 6 3 ) 2 )` yields 4
  - `ABS SQRT REMAINDER MIN MAX`
    - `(abs -7)` yields 7
  - `EQ?`
    - `(eq? 1 1 )` yields `#t`

Special Forms

- Definition

  (define <var> <expr>)

- Conditional

  (if <test> <then> <else>)

```
(if   (zero? 5)   0   1)
```

Data Types

- Data Types
  - Numbers
  - Booleans
  - Strings

- Built-in function return type
  - Numbers
    - `+, -, *, =  : number?`
  - Booleans
    - `#t, #f  : boolean?`
  - Strings
    - `string->list  : string?`

Quoted symbol

- `QUOTE` - takes one parameter; returns the parameter without evaluation
  - `QUOTE` is used to avoid parameter evaluation
  - `QUOTE` is needed because arguments passed to functions are evaluated before they are passed.
  - `QUOTE` can be abbreviated with the apostrophe prefix operator
    - Eg)
      'A is equivalent to (QUOTE A)
      '(A B) is equivalent to (QUOTE (A B))

Lists

- A list consists of space-separated objects between parenthesis, and like a symbol, must be quoted.
  - Without quote, (A B C) means
    - evaluate A to a function, evaluate B and C as parameters

```
'(1 2 3)
'(list of symbols)
'( (a b) (c d) )
```

- () represents the empty list
- operations
  - `car, cdr, cons, null?, ...`
  - `list, append, ...`
  - `first, second, ..., ninth`

List Functions: CONS and LIST

- `CONS` takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result

  e.g., `(CONS 'A '(B C))` returns `(A B C)`

- `LIST` takes any number of parameters; returns a list with the parameters as elements

Creating Lists with list

- LIST takes all of its arguments and packs them into a list.

```
1 ]=> (list 1 2 3 4 )

;Value 1: (1 2 3 4 )
```

- (A) represents list contains A
  - (A) is really (A . ())
- (A B) is really (A . (B . ()))

CONS

- CONS returns a newly created pair

```
1 ]=> (cons 'a 'b)
;Value 1: (a . b)
```

- CONS insert the first arg into the second list

```
1 ]=> (cons 1 (list 2 3 ) )
;Value 1: (1 2 3)

1 ]=> (cons 3 ())
;Value 1: (3)

1 ]=> (cons (list 1 2) (list 3 4))
;Value 1: ((1 2) 3 4)
```

List Functions: CAR and CDR

- `CAR` takes a list parameter; returns the first element of that list

  e.g., `(CAR '(A B C))` yields `A`

  `(CAR '((A B) C D))` yields `(A B)`

- `CDR` takes a list parameter; returns the list after removing its first element

  e.g., `(CDR '(A B C))` yields `(B C)`

  `(CDR '((A B) C D))` yields `(C D)`

CAR, CDR, APPEND Examples

```
1 ]=> (car (list 1 2 3))

;Value: 1

1 ]=> (cdr (list 1 2 3))

;Value: (2 3)
```

- APPEND merges two lists

```
1 ]=> (append (list 1 2 3) (list 4 5) )

;Value: (1 2 3 4 5)
```

Excercise

```
1 ]=> (list (car (list 1 2 3)) (list 9 8))

;Value 1: (1 (9 8))
```

```
1 ]=> (cons (car (list 1 2 3)) (list 9 8))

;Value 1: (1 9 8)
```

```
1 ]=> (car(cdr(cdr (list 1 2 3 4 5))))

;Value 1: 3
```

CADR, CADDR, CADDDR, ….

- `CADR` takes a list parameter; returns the second element of that list
- `CADDR` takes a list parameter; returns the third element of that list
- `CADDDR` takes a list parameter; returns the fourth element of that list
- …

Function Definition: LAMBDA

- Lambda Expressions
  - Form is based on λ notation

  e.g., `(LAMBDA (x) (* x x))`

  `x` is called a bound variable
- Lambda expressions can be applied as follows:

  e.g., `((LAMBDA (x) (* x x)) 7)`

```
$scheme
1 ]=> ((LAMBDA (x) (* x x)) 7)

;Value: 49
```

Special Form Function: DEFINE

- A Function for Constructing Functions `DEFINE`
  1. To bind a symbol to an expression
     e.g., `(DEFINE pi 3.141593)`
     Example use: `(DEFINE two_pi (* 2 pi))`
  2. To bind names to lambda expressions
     e.g., `(DEFINE (square x) (* x x))`

```
1 ]=> (DEFINE (square x) (* x x))

;Value: square

1 ]=> (square 5)

;Value: 25
```

Predicate Function: EQ?

- `EQ?` takes two parameters
- it returns `#T` if both parameters are atoms and the two are the same; otherwise `#F`
  - e.g., `(EQ? 'A 'A)` yields `#T`
    `(EQ? 'A 'B)` yields `#F`
  - `EQ?` does not work for numeric atoms
  - `EQ?` can be called with list parameters, but not recommended.

Equivalence Test

- (eq? (cons 3 ()) (cons 3 ()))
  - #f
- (define a (cons 3 ()))
- (define b (cons 3 ()))
- (eq? a b)
  - #f

Shallow equality test

- (define c a)
- (eq? a c)
  - #t

Numeric Predicate Functions

- `#T` is true and `#F` is false (sometimes `()` is used for false)
- `=, <>, >, <, >=, <=`
- `EVEN?, ODD?, ZERO?, NEGATIVE?`

Control Flow: COND

- Multiple Selection - the special form, `COND`

  General form:

  (**COND**

  (*predicate_1* { *expr* })

  (*predicate_2* { *expr* })

  . . .

  (*predicate_N* { *expr* })

  (**ELSE** { *expr* }))

- Returns the value of the last expression in the first pair whose predicate evaluates to true

Example of COND

```
(DEFINE (compare x y)
   (COND
      ((> x y) "x is greater than y")
      ((< x y) "y is greater than x")
      (ELSE "x and y are equal")
   )
)
```

LIST? and NULL?

- `LIST?` takes one parameter; it returns `#T` if the parameter is a list; otherwise `#F`

- `NULL?` takes one parameter; it returns `#T` if the parameter is the empty list; otherwise `#F`
  - Note that `NULL?` returns `#T` if the parameter is `()`

Example Scheme Function: member

- `member` takes an atom and a simple list; returns `#T` if the atom is in the list; `#F` otherwise

```
DEFINE (member atm lis)
      (COND
            ((NULL? lis) #F)
            ((EQ? atm (CAR lis)) #T)
            ((ELSE (member atm (CDR lis)))
      ))
```

Example Scheme Function: equalsimp

- `equalsimp` takes two simple lists as parameters; returns `#T` if the two simple lists are equal; `#F` otherwise

```
(DEFINE (equalsimp lis1 lis2)
      (COND
            ((NULL? lis1) (NULL? lis2))
            ((NULL? lis2) #F)
            ((EQ? (CAR lis1) (CAR lis2))
                  (equalsimp(CDR lis1)(CDR
lis2)))
            (ELSE #F)
      ))
```

Example Scheme Function: equal

- `equal` takes two general lists as parameters; returns `#T` if the two lists are equal; `#F` otherwise

```
(DEFINE (equal lis1 lis2)
  (COND
     ((NOT (LIST? lis1))(EQ? lis1 lis2))
     ((NOT (LIST? lis2)) #F)
     ((NULL? lis1) (NULL? lis2))
     ((NULL? lis2) #F)
     ((equal (CAR lis1) (CAR lis2))
          (equal (CDR lis1) (CDR lis2)))
     (ELSE #F)
))
```

Example Scheme Function: append

- `append` takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

```
(DEFINE (append lis1 lis2)
  (COND
     ((NULL? lis1) lis2)
     (ELSE (CONS (CAR lis1)
                 (append (CDR lis1) lis2)))
))
```

Example Scheme Function: bstsearch

- Binary search tree example:

```
(define (bstsearch tree value)
  (cond
  ((null? tree) #f)
  ((< value (car tree))
      (bstsearch  (cadr tree) value))
  ((> value (car tree))
      (bstsearch (caddr tree) value))
  ((= value (car tree))
      #t)
  ))

(bstsearch '(7 (4 () (5 () ())) (9 (8 () ()) ())) 4)
```

Tail Recursion in Scheme

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- Scheme language definition requires that Scheme language systems convert all tail recursive functions to use iteration

Tail Recursion in Scheme

- Example of rewriting a function to make it tail recursive, using helper a function

Original:

```
(DEFINE (factorial n)
          (IF (= n 0)
               1
               (* n (factorial (- n 1)))
          )
)
```

Tail recursive:

```
(DEFINE (facthelper n p)
           (IF (= n 0)
                 p
                 facthelper((- n 1) (* n p)))
           )
)

(DEFINE (factorial n) (facthelper n 1))
```

Higher Order functions

- Higher Order functions – take a function as a parameter or returns a function as a result.

Example: the map function

```
(map function list)
(map number? '(1 2 f 8 k))
(map (lambda (x) (+ 5 x)) '(4 5 6 7))
```

Map function in Scheme

- Map transforms a list by applying a function to each of its elements. Its return value is the transformed list.

```
(define (double x) (* 2 x))
(map double (list 1 2 3 4 5))
   => (2 4 6 8 10)
```

- Implement your own Map function
  - Applies the given function to all elements of the given list;

```
(DEFINE (mapcar fun lis)
      (COND
        ((NULL? lis) ())
        (ELSE (CONS (fun (CAR lis))
                    (mapcar fun (CDR lis))))
))
```

Functions That Build Code

- It is possible in Scheme to define a function that builds Scheme code and requests its interpretation
- This is possible because the interpreter is a user-available function, `EVAL`

- Eval takes a Scheme object and an environment, and evaluates the Scheme object.

```
(define x 3) (define y (list '+ x 5))
(eval y user-initial-environment)
```

Adding a List of Numbers

```
((DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+ lis) USER-INITIAL-ENVIRONMENT))
))
```

The parameter is a list of numbers to be added; `adder` inserts a + operator and evaluates the resulting list

- Use `CONS` to insert the atom + into the list of numbers.
- Be sure that + is quoted to prevent evaluation
- Submit the new list to `EVAL` for evaluation

Apply

- The apply function applies a function to a list of its arguments.

Examples:

```
(apply factorial '(3))
(apply + '(1 2 3 4))
```

Functional vs Imperative PL

- Imperative Languages:
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed

- Functional Languages:
  - Simple semantics
  - Simple syntax
  - Inefficient execution
  - Programs can automatically be made concurrent